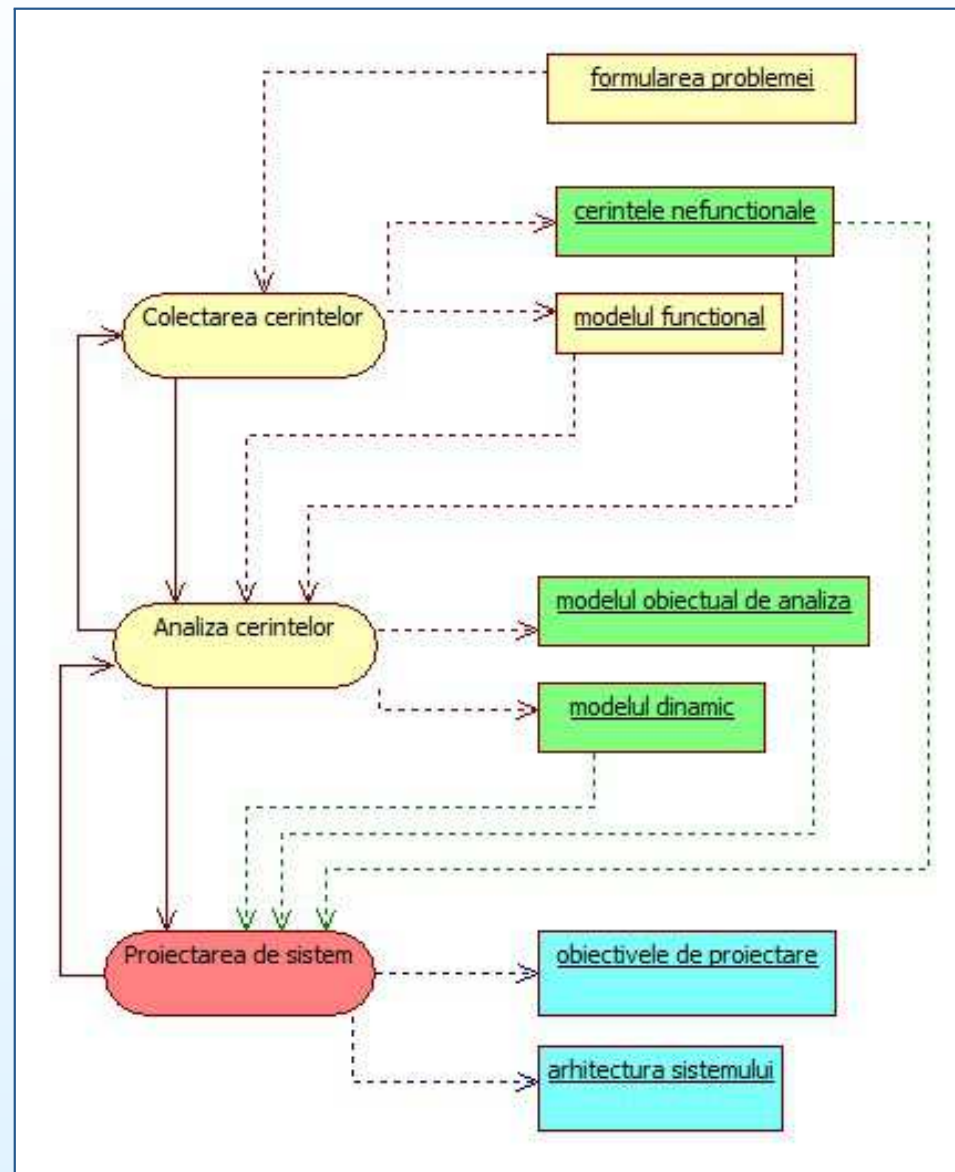


*Curs 5*  
*Proiectarea de sistem (I)*

*Curs bazat pe B. Bruegge and A.H. Dutoit*  
*"Object-Oriented Software Engineering using UML, Patterns, and Java"*

# Proiectarea de sistem



## Proiectarea de sistem (cont.)

---

- Procesul de transformare a modelului rezultat din ingineria cerințelor într-un model arhitectural al sistemului
- Produse ale proiectării de sistem
  - *Obiectivele de proiectare* (eng. *design goals*)
    - Calități ale sistemului pe care dezvoltatorii trebuie să le optimizeze
    - Derivate din cerințele nefuncționale
  - *Arhitectura sistemului*
    - Subsistemele componente (de dimensiuni mai mici, asignabile unei subechipe de dezvoltare)
    - Responsabilitățile subsistemelor și dependențele între ele
    - Maparea subsistemelor la hardware
    - Strategii de dezvoltare: fluxul global de control, strategia de gestionare a datelor cu caracter persistent, politica de control a accesului

# Proiectarea de sistem (cont.)

---

- Activități ale proiectării de sistem
  - *Identificarea obiectivelor de proiectare*
    - Identificarea și prioritizarea acelor calități ale sistemului pe care dezvoltatorii trebuie să le optimizeze
  - *Descompunerea inițială a sistemului*
    - Pe baza modelului funcțional și a modelelor de analiză
    - Bazată pe utilizarea unor stiluri arhitecturale standard
  - *Rafinarea descompunerii inițiale în vederea atingerii obiectivelor de proiectare*
    - Rafinarea arhitecturii de la pasul anterior, până la îndeplinirea tuturor obiectivelor de proiectare
- Analogie cu proiectarea arhitecturală a unei clădiri
  - Componente: camere vs. subsisteme
  - Interfețe: pereți/uși vs. servicii
  - Reproiectare: mutarea pereților vs. schimbarea subsistemelor/interfețelor

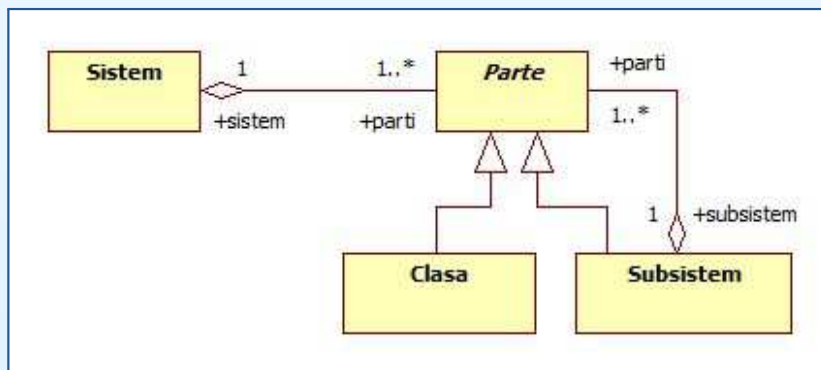
# Concepte în proiectarea de sistem

---

- Subsisteme și clase
- Servicii, interfețe și API-uri
- Coeziune și cuplare
- Stratificare și partiționare
- Stiluri arhitecturale și arhitecturi software

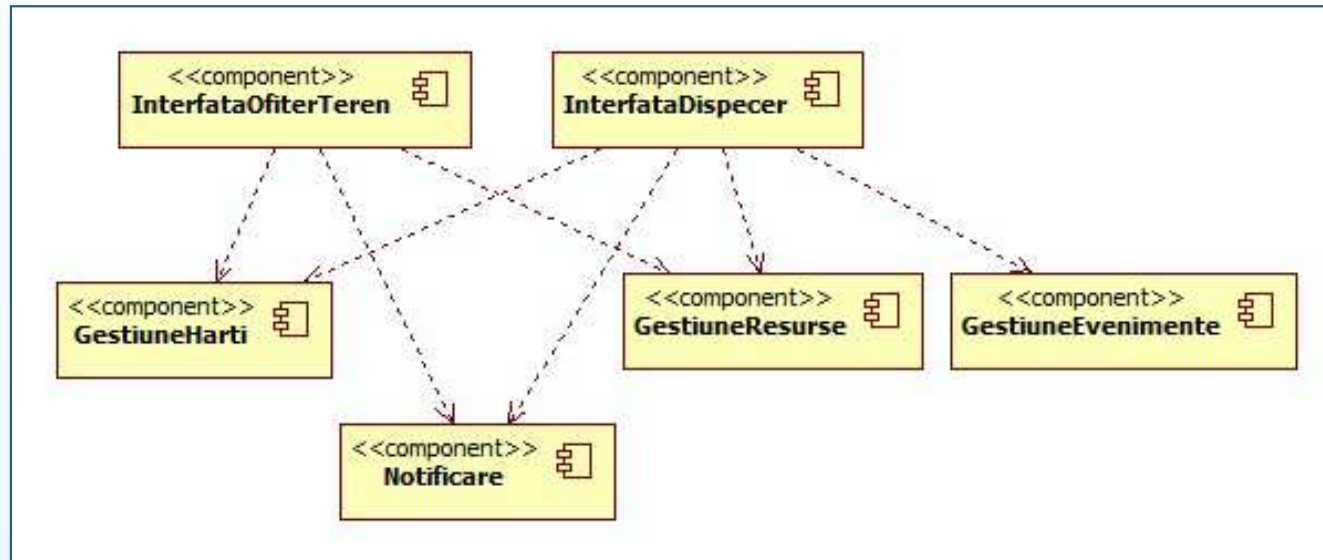
# Subsisteme și clase

- *Subsistem* = parte înlocuibilă a unui sistem (constând într-un număr de clase din domeniul soluției), caracterizată prin interfețe bine definite, care încapsulează starea și comportamentul claselor componente
  - Descompunerea în subsisteme permite gestionarea complexității ("divide et impera")
  - Un subsistem se dezvoltă, de regulă, de către un programator sau o echipă de dezvoltare
  - Prin descompunerea sistemului în sisteme (relativ) independente, se permite dezvoltarea (relativ) concurentă a acestora
  - Sistemelor complexe le corespund mai multe nivele de descompunere (*Composite pattern*)



## Subsisteme și clase (cont.)

- Ex.: descompunere în subsisteme a sistemului SGA (diagramă UML de componente)



- Subsistemele sunt reprezentate ca și componente UML, cu relații de dependență între ele
- O componentă UML poate reprezenta
  - O componenta logică = un subsistem ce nu are un echivalent runtime
  - O componenta fizică = un subsistem ce are un echivalent runtime

# Servicii, interfețe și API-uri

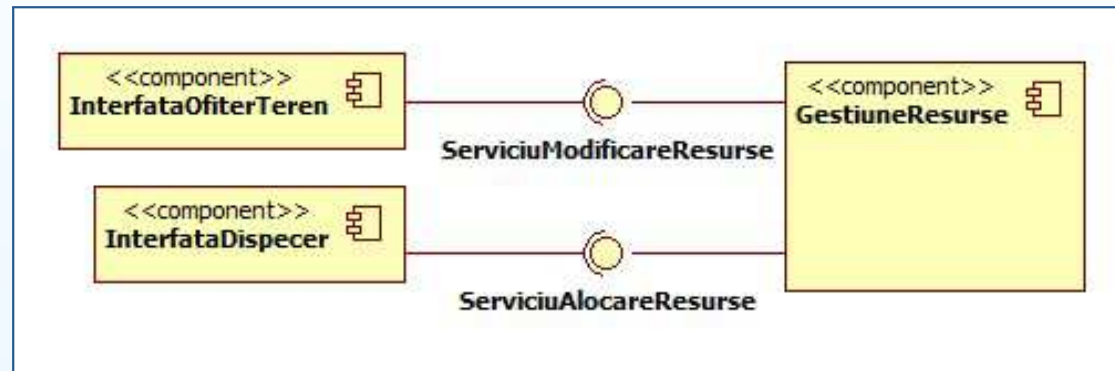
---

- *Serviciu* = mulțime de operații înrudite (definite cu același scop)
  - Subsistemele sunt caracterizate de serviciile oferite altor subsisteme
  - Ex.: un subsistem care oferă un serviciu de notificare va defini operații de tipul *LookupChanel()*, *SubscribeToChanel()*, *UnsubscribeFromChanel()*, *SendNotification()*
  - Serviciile se identifică în timpul proiectării de sistem
- *Interfață* (a unui subsistem) = mulțime de operații UML înrudite, complet tipizate (numele, parametrii + tipurile lor, tipul returnat)
  - Rafinare a unui serviciu, specifică interacțiunile și fluxul de informații dinspre și înspre frontiera subsistemului (nu și în interiorul acestuia)
  - Interfețele se definesc în timpul proiectării obiectuale
- *API (Application Programming Interface)* = specificare a unei interfețe subsistem într-un limbaj de programare
  - API-urile se definesc în etapa de implementare



## Servicii, interfețe și API-uri (cont.)

- Ex.: Interfețe/servicii oferite (eng. *provided*) și solicitate/utilizate (eng. *required*)



- Notația UML: conectori *ball-and-socket*
  - Ball (lollipop) = interfață oferită, socket = interfață solicitată
  - Dependențele dintre subsisteme se reprezintă prin cuplarea conectorilor ball cu cei socket
  - Reprezentare utilizată în momentul în care descompunerea în subsisteme e relativ stabilă și focusul se schimbă de pe identificarea subsistemelor pe identificarea serviciilor (anterior se folosesc relații UML de dependență)

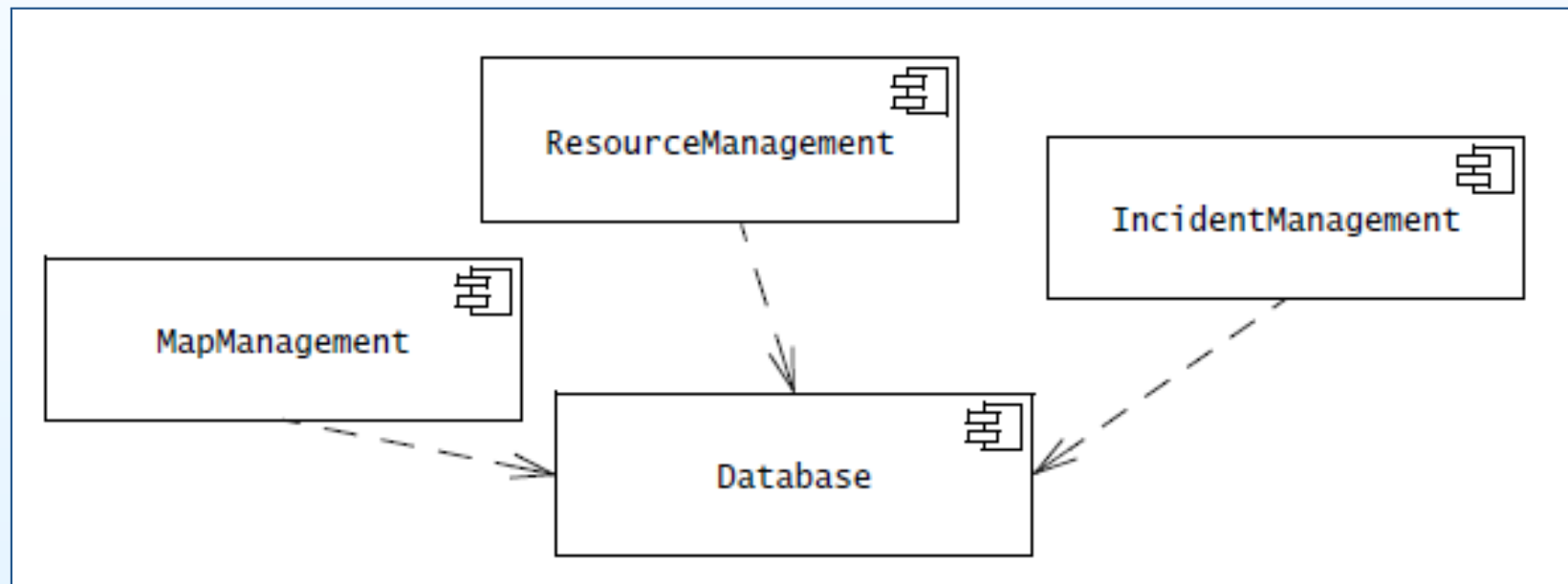
# Coeziune și cuplare

---

- *Cuplare* (eng. *coupling*) = măsură a dependenței dintre două subsisteme
  - *Cuplare slabă* (eng. *low coupling*) - număr mic de dependențe (subsisteme relativ independente)
  - *Cuplare strânsă* (eng. *strong coupling*) - număr mare de dependențe (schimbările efectuate într-un sistem îl vor afecta și pe celălalt)
  - Dezirabilă într-o descompunere: cuplarea slabă
  - Reducerea cuplării conduce, în general, la creșterea complexității prin introducerea de subsisteme suplimentare
- *Coeziune* (eng. *coesion*) = măsură a dependențelor din interiorul unui subsistem
  - *Coeziune înaltă* (eng. *high coesion*) - subsistemul conține un număr mare de clase puternic relaționate și care efectuează sarcini similare
  - *Coeziune slabă* (eng. *low coesion*) - subsistemul conține un număr de clase nerelaționate
  - Dezirabilă într-o descompunere: coeziunea înaltă
  - Creșterea coeziunii (prin descompunere în subsisteme cu coeziune înaltă) conduce și la creșterea cuplării, prin dependențele nou introduse

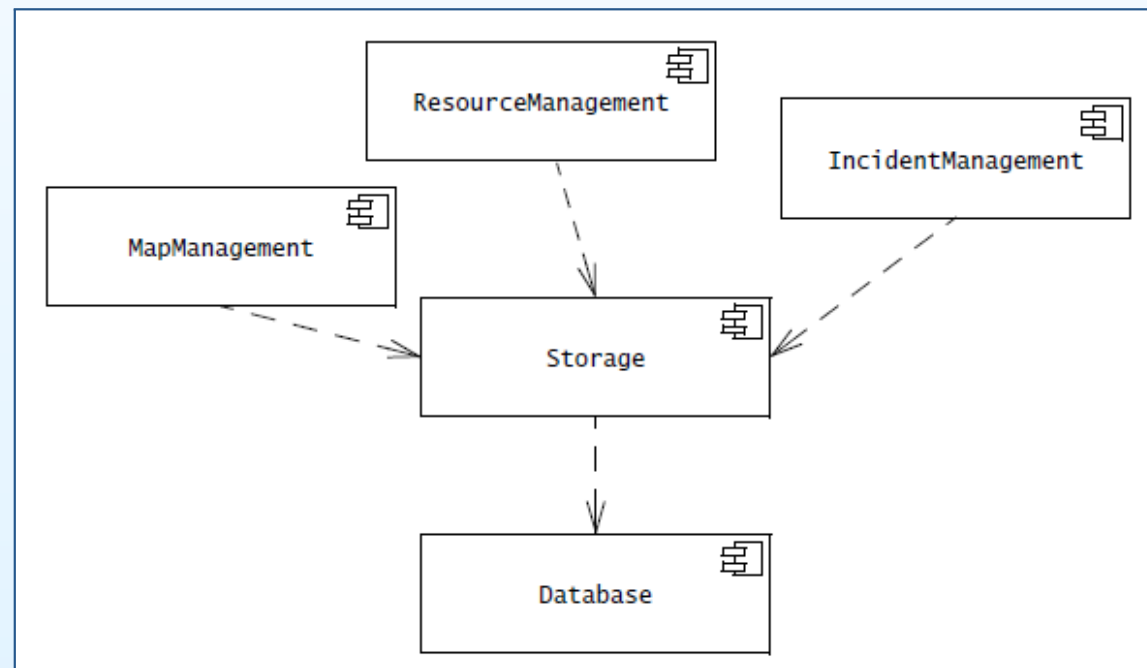
# Cuplare

- Ex.: Subsisteme stâns cuplate
  - Subsistemele de gestiune trimit SGBD-ului comenzi SQL
  - Trecerea la un alt SGBD sau schimbarea strategiei de persistență (fișiere text) determină modificări la nivelul tuturor celor trei subsisteme de gestiune



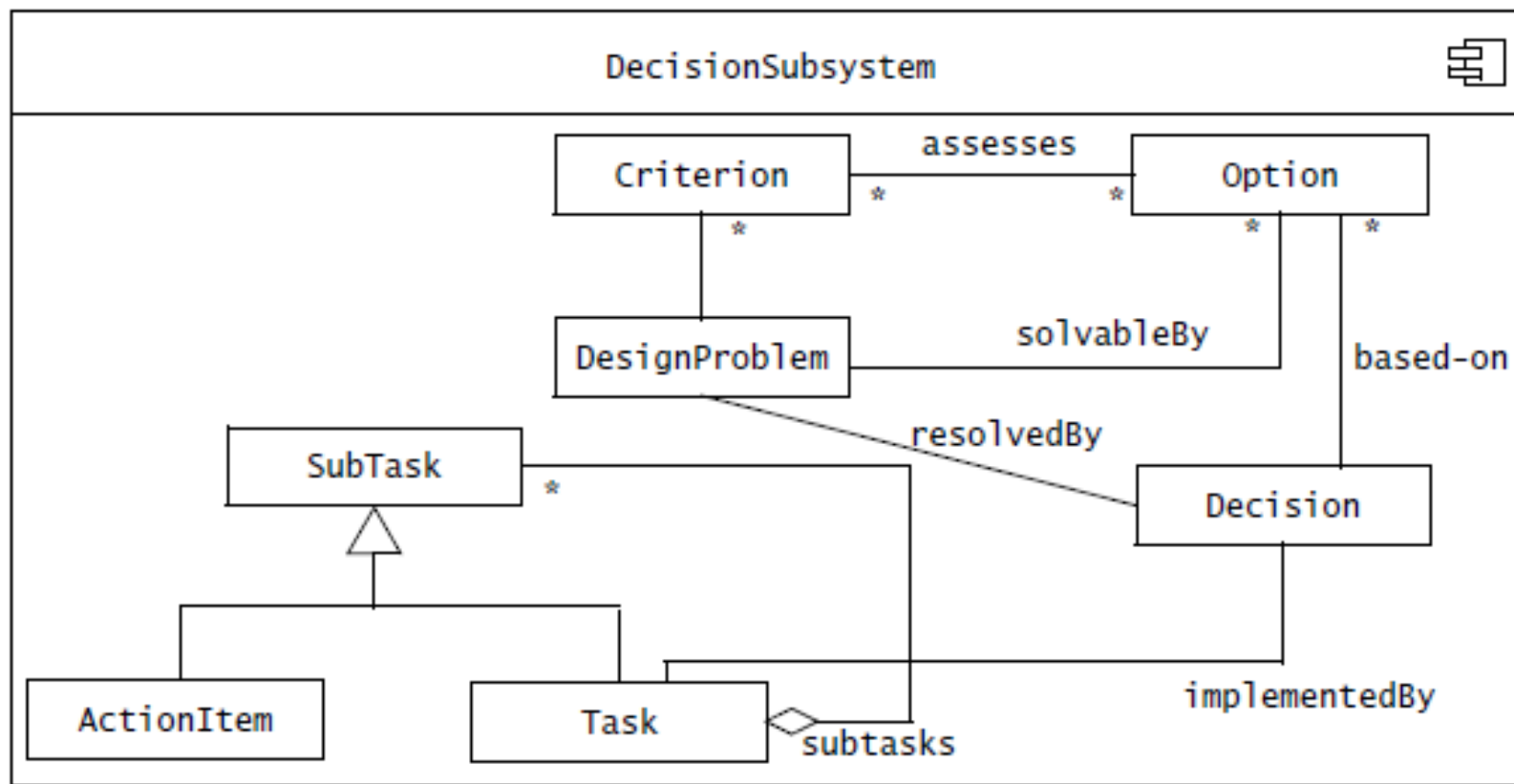
## Cuplare (cont.)

- Ex.: Reducerea cuplării prin inserarea unui subsistem suplimentar
  - Noul subsistem izolează subsistemele de gestiune de SGBD
  - Subsistemele de gestiune utilizează doar serviciile oferite de noul subsistem, care va fi responsabil cu trimiterea de comenzi SQL spre SGBD
  - Trecerea la un alt SGBD sau schimbarea strategiei de persistență va determina doar modificări la nivelul subsistemului introdus



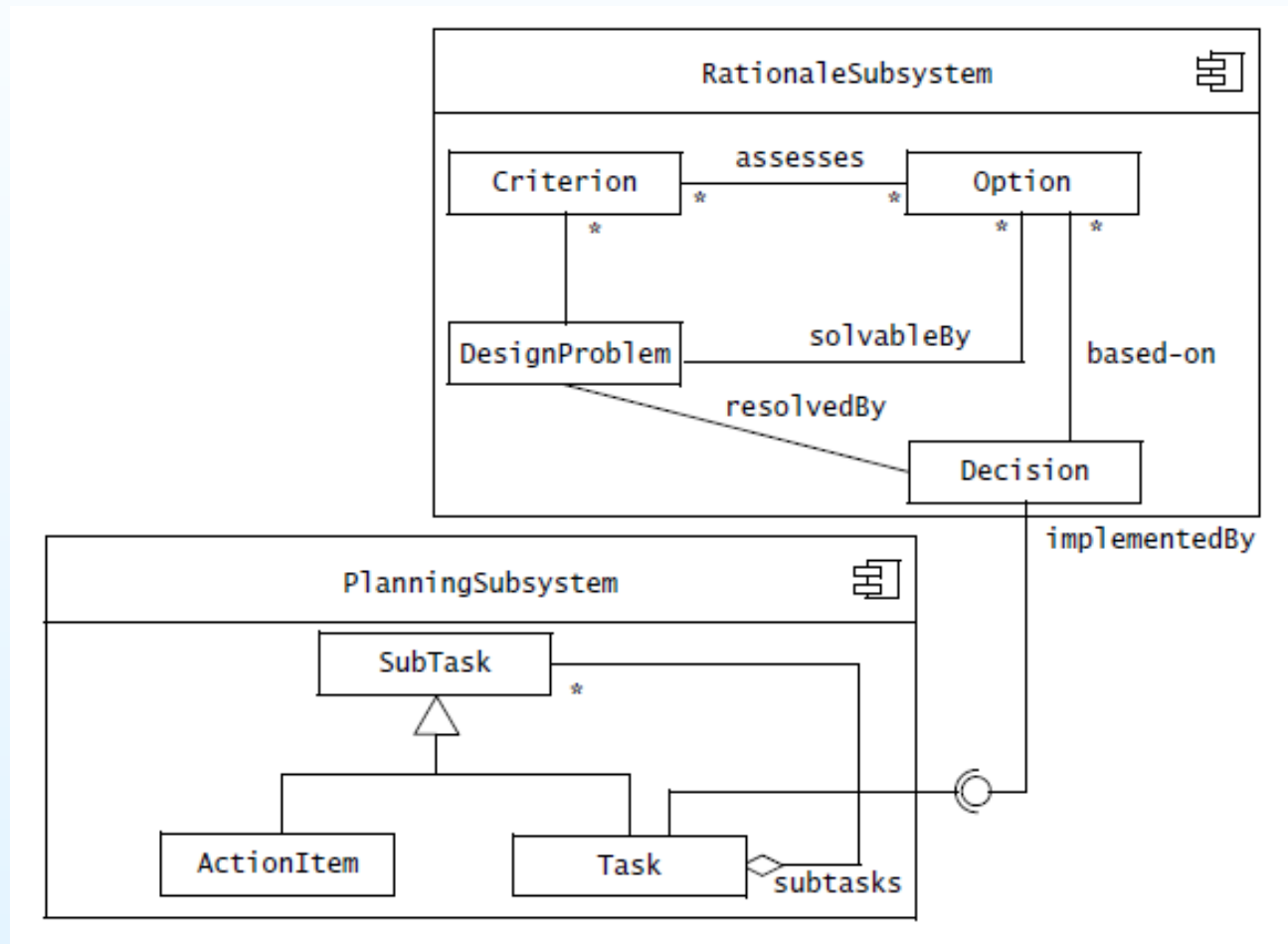
# Coeziune

- Subsistem cu coeziune slabă
  - Clasele componente pot fi partiționate în două submulțimi slab cuplate



## Coeziune (cont.)

- Creșterea coeziunii prin descompunerea subsistemului inițial



# Stratificare și partiționare

---

- *Descompunere ierarhică a unui sistem (stratificare)*
  - Generează o mulțime ordonată de *straturi* (eng. *layers*)
  - Un strat reprezintă un grup de subsisteme ce oferă servicii înrudite, eventual utilizând servicii dintr-un alt strat
  - Straturile sunt ordonate: un strat poate accesa doar servicii ale straturilor inferioare
    - *Arhitectură închisă* - fiecare strat poate accesa doar servicii din stratul imediat inferior (scop = modificabilitate/flexibilitate)
    - *Arhitectură deschisă* - un strat poate accesa servicii din oricare dintre straturile inferioare (scop = eficiență )
- *Partiționare*
  - Generează un grup de subsisteme la același nivel (eng. *peers*), fiecare dintre ele fiind responsabil de o categorie diferită de servicii
- În general, o descompunere a unui sistem complex este rezultatul atât al stratificării, cât și al partiționării

# Stiluri arhitecturale și arhitecturi software

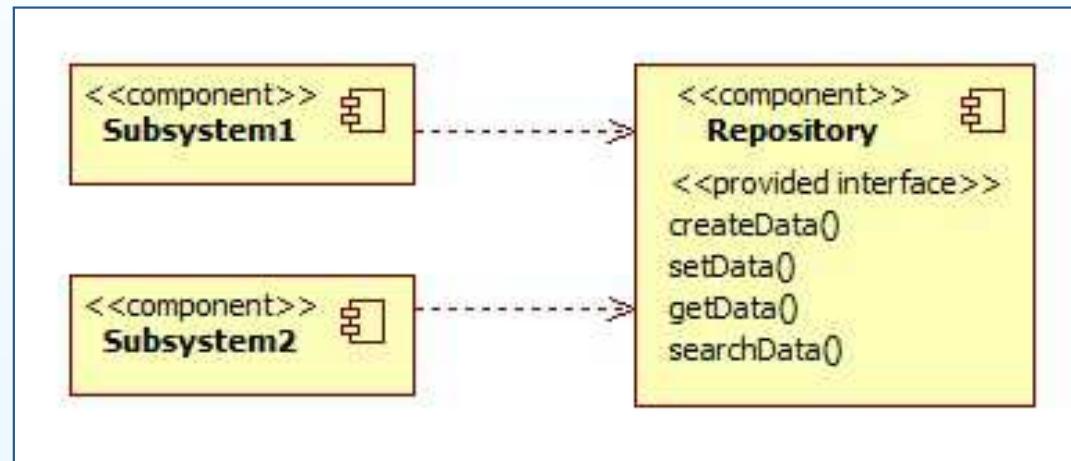
---

- *Descompunerea sistemului* (eng. *system decomposition*) = identificarea subsistemelor, a serviciilor și a relațiilor între acestea
- *Stil arhitectural* (eng. *architectural style*) = șablon de descompunere a sistemelor
- *Arhitectură software* (eng. *software architecture*) = instanță a unui stil arhitectural
- Exemple de stiluri arhitecturale
  - Repository
  - Model-View-Controller
  - Client-Server
  - Peer-to-Peer
  - Three-tier architecture
  - Four-tier architecture
  - Pipes and filters



# Repository

- Subsistemele accesează și modifică o singură structură de date centralizată, denumită *repository*



- Subsistemele sunt relativ independente și interacționează doar prin intermediul repository-ului (cuplare slabă)
- Fluxul de control poate fi dictat de repository (prin triggere) sau de către subsisteme (prin blocaje și sincronizări)

# Repository (cont.)

---

- Avantaje

- Arhitectură utilă în cazul sistemelor cu necesități de procesare complexe, în continuă schimbare
- Odată definit repository-ul, pot fi oferite servicii noi prin definirea de subsisteme adiționale

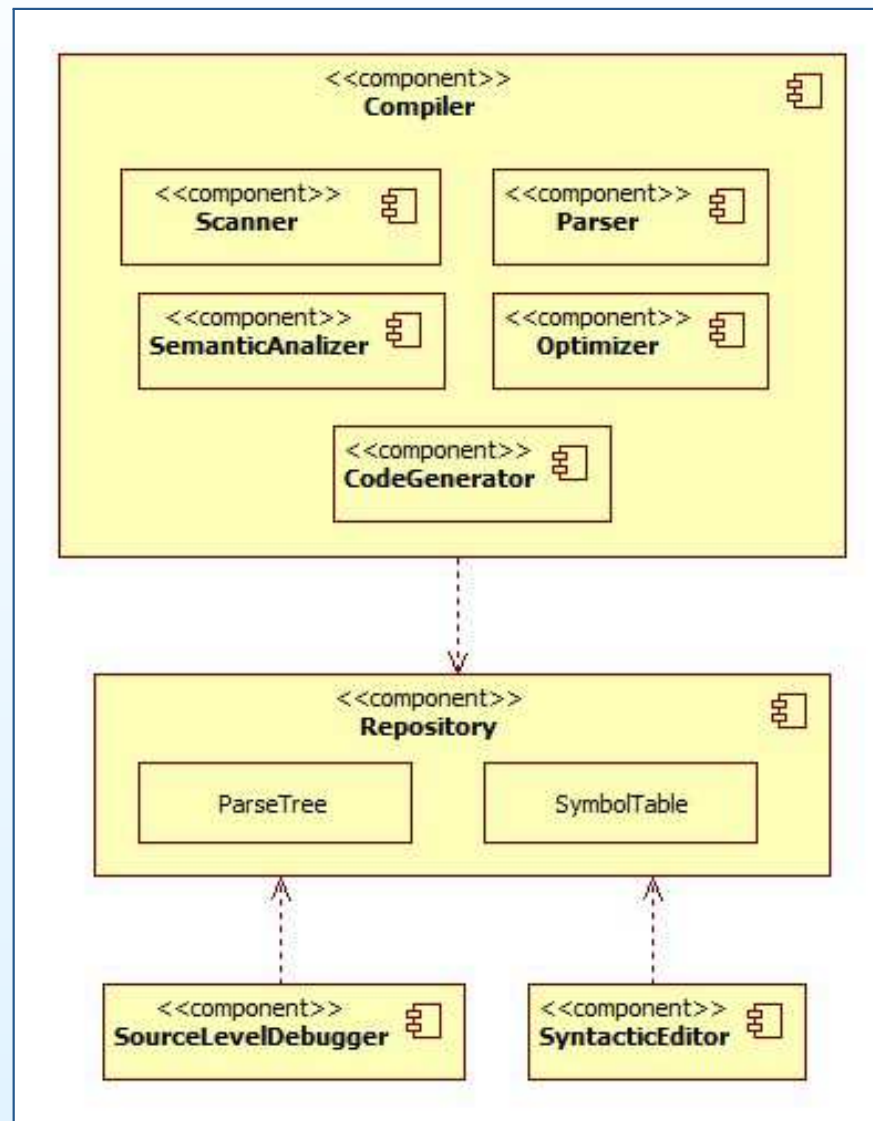
- Dezavantaje

- Subsistemele și repository-ul sunt strâns cuplate, făcând dificilă modificarea repository-ului fără a afecta subsistemele
- Probleme de performanță

- Utilitate

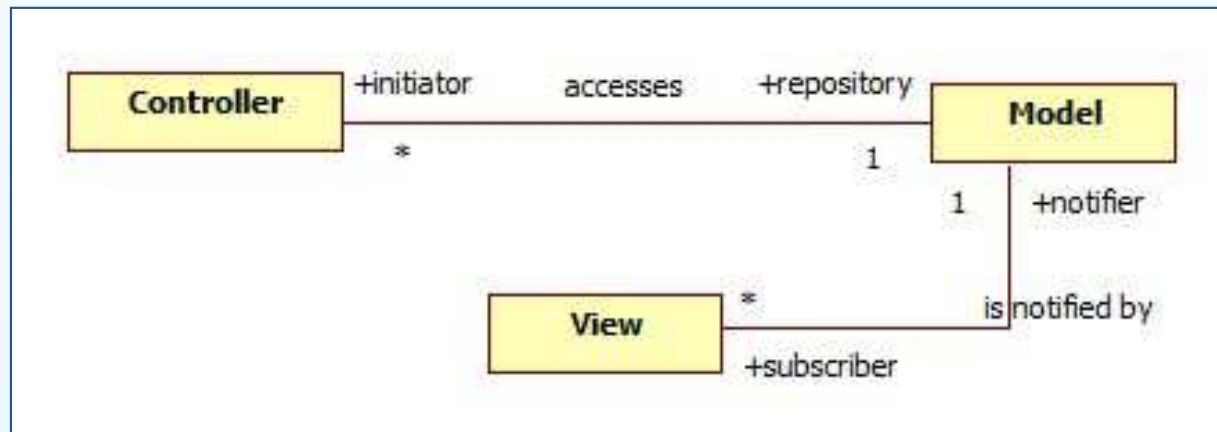
- Sisteme de gestiune a bazelor de date
- Compilatoare și medii integrate de dezvoltare (eng. *Integrated Development Environments, IDEs*)

# Exemplu de arhitectură Repository pentru un IDE



# Model-View-Controller (MVC)

- Subsistemele sunt încadrate în una din trei categorii
  - *Model* - reprezintă informații/cunoștințe din domeniul problemei
  - *View* - afisează aceste informații utilizatorului
  - *Controller* - translatează interacțiunile cu *view*-ul în acțiuni asupra modelului



- Subsistemele model nu depind de nici un subsistem view sau controller
  - Modificările produse la nivelul acestora sunt propagate în subsistemele view prin intermediul unui protocol de înscriere/ notificare
  - Funcționalitatea de înscriere/notificare este realizată, de obicei, cu ajutorul șablonului de proiectare *Observer*

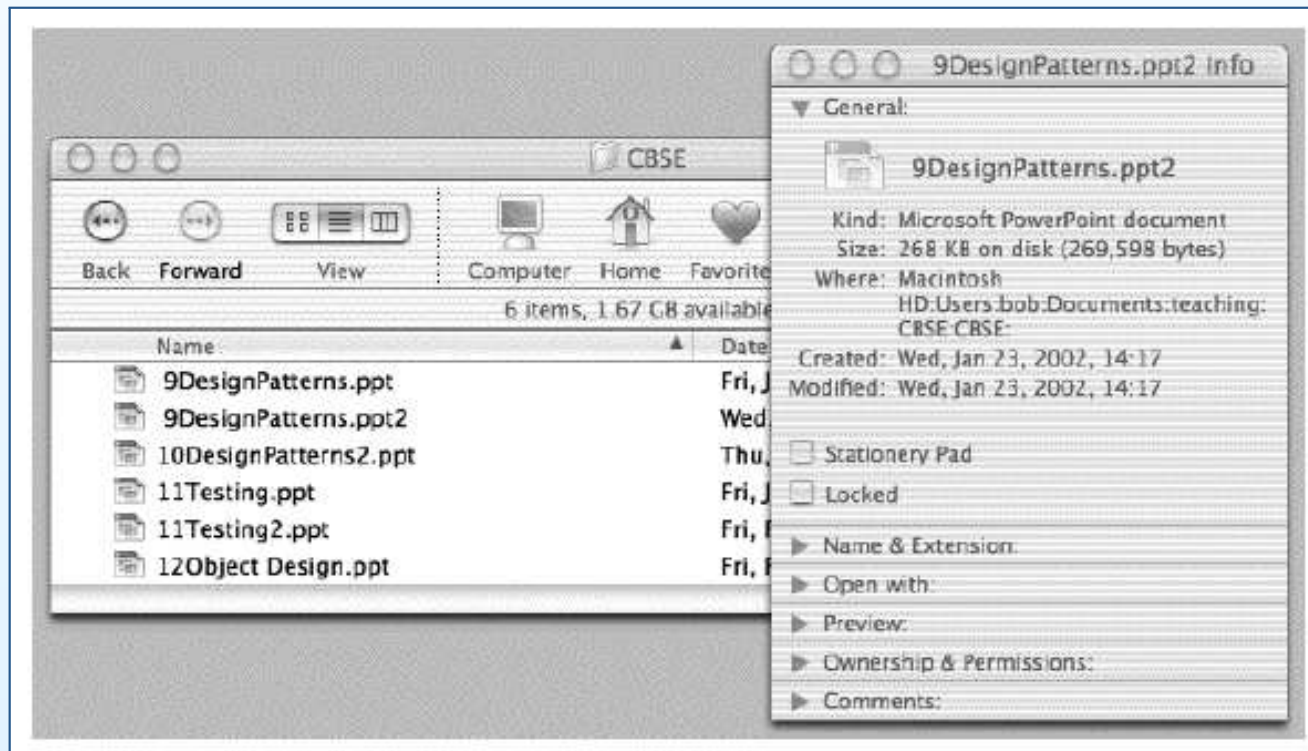
## Model-View-Controller (cont.)

---

- Justificare
  - Interfața utilizator (view-urile și controller-ele) sunt mult mai predispuse spre schimbare decât informațiile din model
  - Pot fi adăugate vederi noi, fără a modifica în alt fel sistemul
- Utilitate
  - Sisteme interactive, mai ales cele care necesită diferite vederi ale aceluiași model
- MVC este un tip particular de Repository, în care modelul corespunde structurii repository, iar fluxul de control este dictat de către obiectele controller

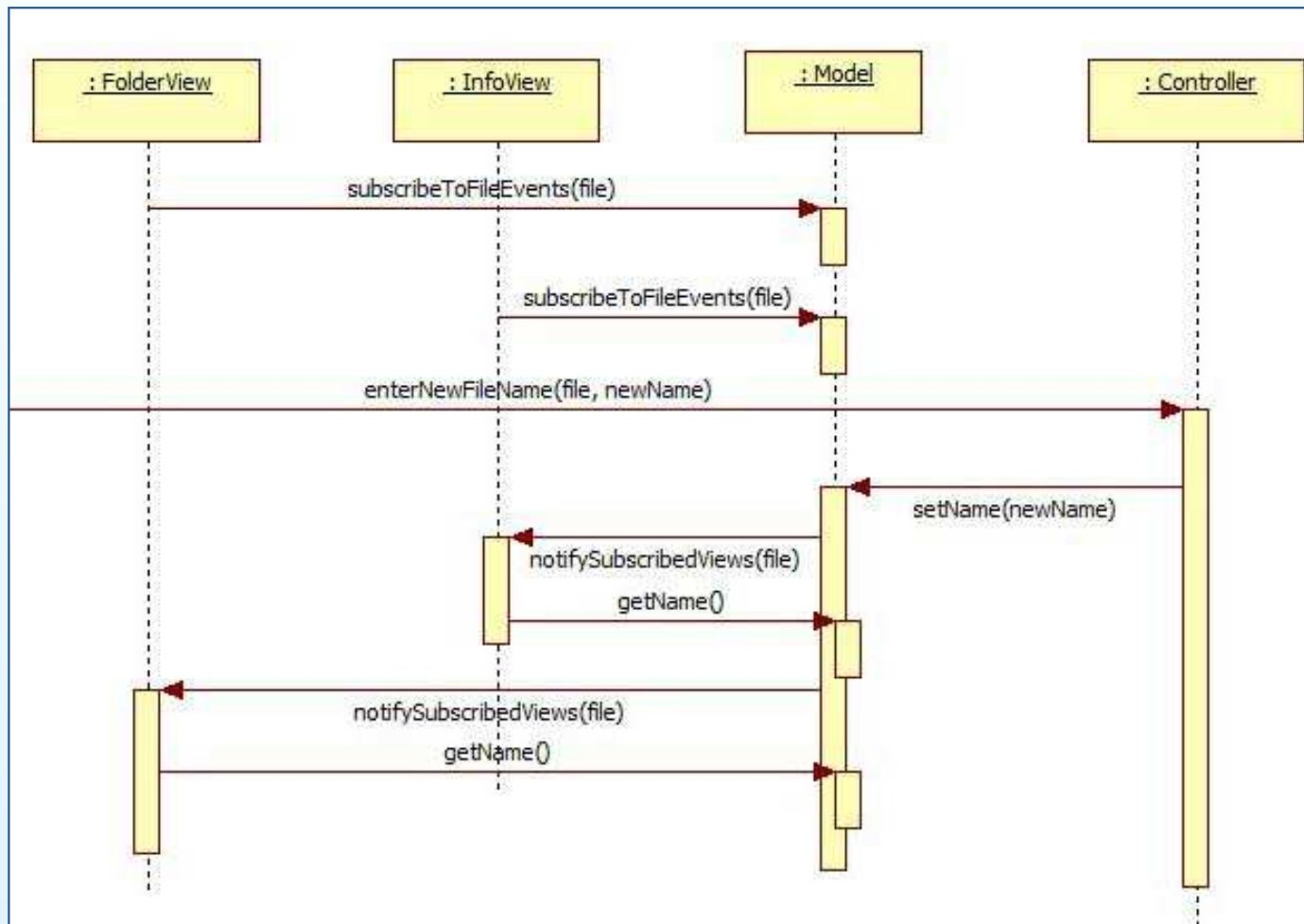
# Exemplu de arhitectură MVC

- Modelul este reprezentat de fișierul *9DesignPatterns.ppt2*
- Una dintre vederi este fereastra CBSE, care listează conținutul directorului cu același nume, cealaltă este fereastra Info, care afișează informații relativ la fișierul *9DesignPatterns.ppt2*
- Schimbarea numelui fișierului determină actualizarea ambelor vederi



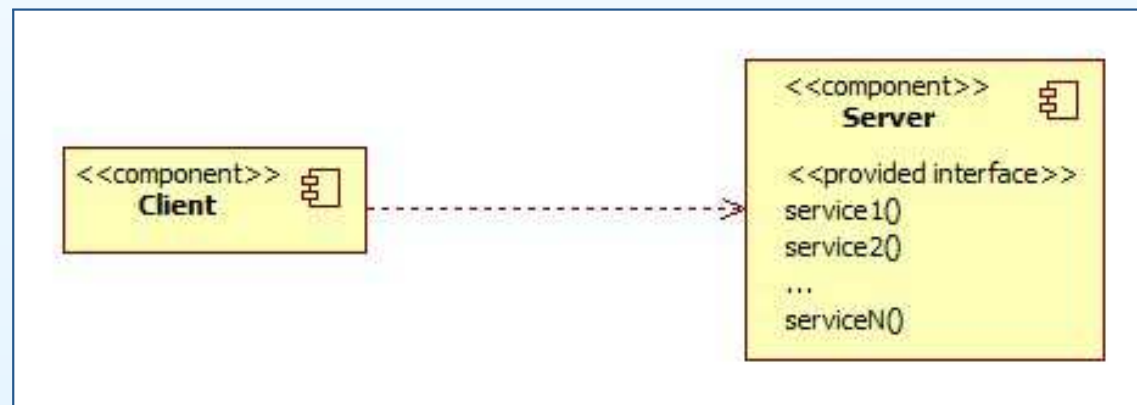
## Exemplu de arhitectură MVC (cont.)

- Secvența de interacțiuni aferentă schimbării numelui fișierului  
*9DesignPatterns.ppt2*



# Client-Server

- Un subsistem, numit *server*, oferă servicii instanțelor unor alte subsisteme, numite *clienți*, care sunt responsabile de interacțiunea cu utilizatorii
  - Solicitarea unui serviciu se face, de obicei, printr-un mecanism de apel la distanță (Java RMI, CORBA, HTTP)
  - Fluxurile de control din server și clienți sunt independente, cu excepția sincronizărilor pentru gestionarea cererilor și primirea răspunsurilor

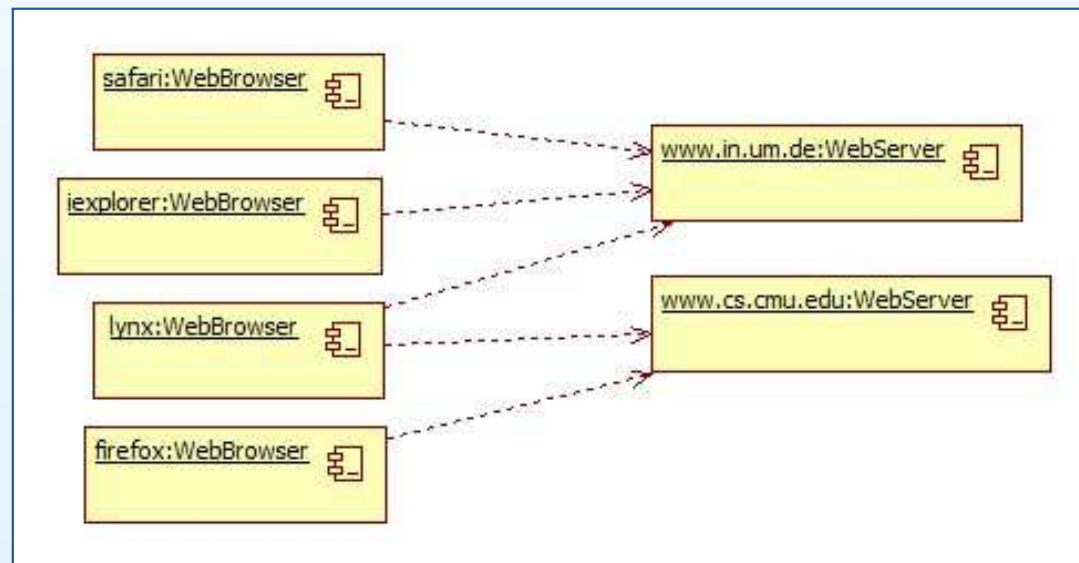


- Utilitate
  - Sisteme distribuite complexe, care gestionează un volum mare de date



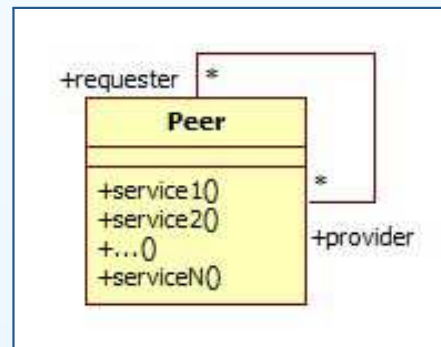
# Exemple de arhitecturi client-server

- Un sistem informațional cu o bază de date centralizată
  - Clienții sunt responsabili de colectarea inputurilor utilizator, validarea acestora și inițierea tranzacțiilor cu baza de date
  - Serverul este responsabil de executarea tranzacțiilor și garantarea integrității datelor
  - În acest caz, stilul client/server este o particularizare a stilului repository, în care structura de date centralizată este gestionată de un proces
- WWW - un client accesează date oferite de diverse servere



# Peer-to-peer

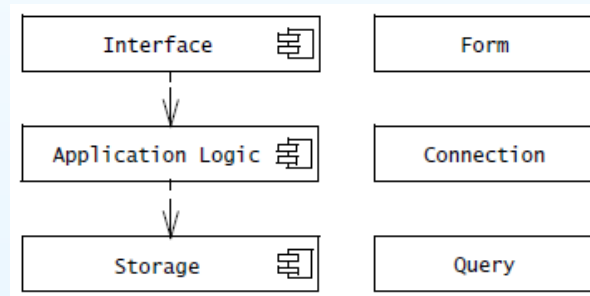
- Generalizare a stilului arhitectural client-server: un subsistem poate juca atât rol de client, cât și de server (fiecare subsistem poate solicita și oferi servicii)
  - Fluxurile de control sunt independente, cu excepția sincronizărilor pe cereri



- Exemple
  - O bază de date care acceptă cereri de la o aplicație, dar o și notifică atunci când se produc schimbări asupra datelor

# Three-tier architecture

- Subsistemele sunt organizate pe trei straturi/nivele
  - *interfață* utilizator - include toate obiectele *boundary* care mediază interacțiunea cu utilizatorii (ferestre, forme, pagini Web, etc.)
  - *logica aplicației* - include toate obiectele *entity* și *control* care realizează verificările, procesările și notificările cerute de aplicație
  - *accesul la date* - gestionează și oferă acces la datele cu caracter persistent



- Avantaje
  - Nivelul de acces la date joacă rolul repository-ului din stilul arhitectural omonim și poate fi partajat de către aplicațiile care operează asupra respectivelor date
  - Separarea dintre logica aplicației și interfață permite modificări ale interfeței fără a afecta logica aplicației

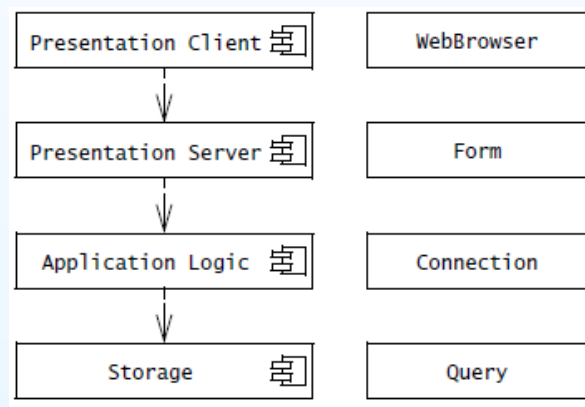
# MVC vs. Three-tier architecture

---

- Stilul arhitectural MVC este nonierarhic (triangular)
  - Subsistemul *view* trimite cereri către subsistemul *controller*
  - Subsistemul *controller* actualizează subsistemul *model*
  - Subsistemul *view* este notificat de către subsistemul *model*
- Stilul arhitectural 3-tier este ierarhic (liniar)
  - Nivelul de prezentare nu comunică niciodată direct cu cel de date (arhitectură închisă)
- MVC nu acoperă problema persistenței datelor

# Four-tier architecture

- O variație a stilului arhitectural three-tier, în care nivelul *interfață* se descompune în
  - *prezentare client* - localizat pe mașinile client
  - *prezentare server* - localizat pe unul sau mai multe servere



- Avantaje
  - Pe nivelul prezentare client pot exista diferiți clienți, o parte a obiectelor *boundary* fiind reutilizate
  - Ex.: un sistem bancar include pe nivelul de prezentare client o interfață Web pentru utilizatori, un ATM și o interfață desktop pentru angajații băncii. Formele partajate de toți clienții sunt definite și procesate la nivelul prezentare server, eliminând redundanța între clienți

# Pipes and filters

---

- *Pipeline* - lanț de unități de procesare (procesare, thread-uri, ...) aranjate astfel încât output-ul uneia reprezintă input-ul următoarei
- *Pipes and filters* - stil arhitectural constând din două tipuri de subsisteme, denumite *pipes* (canale) și *filters* (filtre)
  - *Filter* - subsistem care efectuează un pas de procesare
  - *Pipe* - conexiune dintre doi pași de procesare
- Fiecare subsistem filtru are un canal de intrare și unul de ieșire
  - Datele preluate din canalul de intrare sunt procesate de către filtru și trimise canalului de ieșire
- Ex. Unix shell: `ls -a | cat`